Abdelrahman Mostafa

Bachelor's Thesis

Development of an automated powder dosing system using a 6-DOF collaborative robotic arm (cobot)

by investigating the influence of vibration, angle of dosing, and rotational speed to the mass flow of the powder

Eng. Abdelrahman Mostafa

Supervised by:
Dr. Rainer Schramm, Fluxana GmbH
Prof. Dr. Ronny Hartanto, HSRW

October 5, 2023

An Awesome Publisher

Abstract

Abdelrahman Mostafa

Acknowledgement

Contents

List of Figures

List of Tables

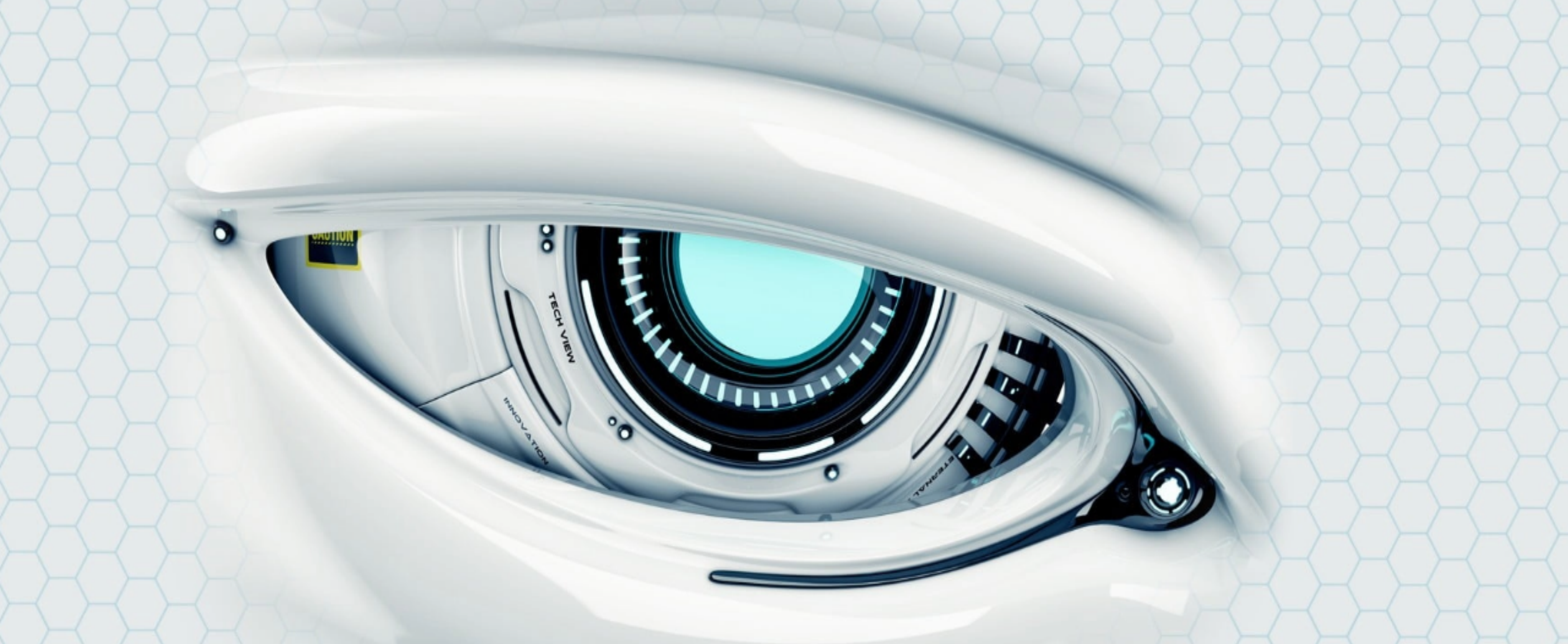
Introduction 1

1.1 Motivation

1.2 Objectives

1.3 Thesis Structure

Background Knowledge, Robotics, ROS, and Powder Dosage



2 Basics of Robotics

As an academic field, robotics emerges as a relatively youthful discipline, characterized by profoundly ambitious objectives, the most paramount of which is the creation of machines capable of emulating human behavior and cognitive processes. This quest to engineer intelligent machines inherently compels us to embark on a journey of self-exploration. It prompts us to scrutinize the intricacies of our own design—why our bodies possess the configurations they do, how our limbs synchronize in movement, and the mechanisms behind our acquisition and execution of intricate tasks. The realization that the fundamental inquiries in robotics are intrinsically linked to inquiries about our own existence forms a captivating and immersive aspect of the robotics pursuit. [8]

This chapter delves into an exploration of various robot classifications, delving into the foundational principles of mechanics and kinematics. It also scrutinizes the intricacies of planning and control within the context of collaborative robots (cobots).

Definition

A robot comprises a mechanical assembly composed of interconnected links, which are joined by diverse types of joints. Typically, these links are depicted as rigid bodies. Additionally, an end-effector, such as a gripper, may be affixed to one of the robot's links. The initiation of robot movement is orchestrated through actuators, which impart forces and torques to the joints, thereby inducing the robot's motion.

2.1 Types of Robots

Across diverse industries, robotics solutions have emerged as catalysts for heightened productivity, elevated safety standards, and increased operational adaptability. Organizations at the vanguard of innovation are discerning forward-looking applications of robotics that yield palpable and quantifiable outcomes. Intel collaborates closely with manufacturers, system integrators, and end-users, actively contributing to the realization of robots that deliver impactful, human-centered results.

According to an article from Intel regarding the classification of robots [5], the current generation of robots has been categorized into six distinct groups.

Autonomous Mobile Robots (AMRs) [4] AMRs navigate their environments and make rapid decisions on the fly. These robots employ advanced technologies like sensors and cameras to gather data from their surroundings. Equipped with onboard processing capabilities, they analyze this data and make well-informed decisions—whether it involves avoiding an approaching human worker, selecting the exact parcel to pick, or determining the suitable surface for disinfection. These robots are self-sufficient mobile solutions that operate with minimal human intervention. [13]

Figure 2.1: The NASA Robonaut 1st generation as a humanoid robot [1] [2]

Automated Guided Vehicles (AGVs) [18] While AMRs navigate their surroundings autonomously, AGVs typically operate along fixed tracks or predetermined paths and frequently necessitate human supervision. AGVs find extensive application in scenarios involving the transportation of materials and goods within controlled settings like warehouses and manufacturing facilities.

Humanoids [7] While numerous mobile humanoid robots could, in a technical sense, be classified as Autonomous Mobile Robots (AMRs), this categorization primarily applies to robots fulfilling human-centric roles, frequently adopting human-like appearances. These robots leverage a similar array of technological components as AMRs to perceive, strategize, and execute tasks, encompassing activities such as offering navigational assistance or providing concierge services.

Hybrids [16] Diverse categories of robots are frequently integrated to engineer hybrid solutions that possess the capacity to execute intricate operations. For instance, the fusion of an AMR with a robotic arm can yield a versatile system tailored for the handling of packages within a warehouse environment. As functionalities are amalgamated within single solutions, there is a concurrent consolidation of computational capabilities.

Articulated Robots [15] Commonly referred to as robotic arms, are designed to replicate the versatile functions of the human arm. These systems typically incorporate a range of rotary joints, varying from two to as many as ten. The inclusion of additional joints or axes equips these robotic arms with a wider range of motion capabilities, rendering them particularly well-suited for tasks such as arc welding, material manipulation, machine operation, and packaging.

Cobots [11] Collaborative Robots, commonly referred to as cobots, are engineered with the specific purpose of working in tandem with, or directly alongside, human operators. Unlike many other categories of robots that function autonomously or within strictly segregated workspaces, cobots share work environments with human personnel to enhance their collective productivity. Their primary role often involves the removal of manual, hazardous, or physically demanding tasks from daily operations. In certain scenarios, cobots are capable of responding to and learning from human movements, further enhancing their adaptability.

The initial four robots fall under the category of mobile robots, possessing the capability to navigate within their surroundings, while the latter two are categorized as stationary robots, as detailed in table 2.1 below.

Within the scope of this paper, our exclusive focus will be on **collaborative robots** [11], commonly referred to as 'cobots'. Since across all the experiments conducted in this study, a cobot has been consistently utilized.

Mobile	Stationary
AMRs AGVs Humanoids Hybrids	Articulated robots Cobots

Table 2.1: Robots Classification.

2.2 Robotic Arm Kinematics

2.3 Robot Control (Software)

Robot Operating System | ROS 3

In this chapter I will describe the most common options used, both the ones inherited from `scrbook` and the `kao`-specific ones. Options passed to the class modifies its default behaviour; beware though that some options may lead to unexpected results...

3.1 Linux for Robotics

Linux is a free, open-source operating system that includes several utilities that will significantly simplify your life as a robot programmer. And as will be shown in the next chapter, ROS (Robot Operating System) is based on a Linux system. All commands and concepts explained here are taken from the Linux tutorial made by the University of Surrey.[6]

3.1.1 What Is Ubuntu? and Why for Robotics?

Ubuntu, accessible at `www.ubuntu.com`, stands as a widely acclaimed Linux distribution rooted in the Debian architecture (source: `https://en.wikipedia.org/wiki/Debian`). Notably, it's freely available and open source, permitting extensive customization for specific applications. Ubuntu boasts an extensive software repository, comprising over 1,000 software components, encompassing essentials such as the Linux kernel, GNOME/KDE desktop environments, and a suite of standard desktop applications, including word processing tools, web browsers, spreadsheets, web servers, programming languages, integrated development environments (IDEs), and even PC games. Versatile in its deployment, Ubuntu can operate on both desktop and server platforms, accommodating architectures like Intel x86, AMD-64, ARMv7, and ARMv8 (ARM64). Canonical Ltd., headquartered in the UK (`www.canonical.com`), provides substantial backing to Ubuntu.

Figure 3.1: Ubuntu

In the realm of robotics, software stands as the nucleus of any robotic system. An operating system serves as the foundation, facilitating seamless interaction with robot actuators and sensors. A Linux-based operating system, such as Ubuntu, offers unparalleled flexibility in interfacing with low-level hardware while affording provisions for tailored OS configurations tailored to specific robot applications. Ubuntu's merits in this context are manifold: it exhibits responsiveness, maintains a lightweight profile, and upholds stringent security measures. Additionally, Ubuntu boasts a robust community support ecosystem and a cadence of frequent releases, ensuring its perpetual relevance. It also offers long-term support (LTS) releases, guaranteeing user assistance for up to five years. These compelling attributes have cemented Ubuntu as the preferred choice among developers in the Robot Operating System (ROS) community. Indeed, Ubuntu stands as the sole operating system that enjoys comprehensive support from ROS developers. The Ubuntu-ROS synergy emerges as the quintessential choice for programming robots.

3.2 Philosophy Behind ROS

The philosophical objectives of ROS can be succinctly described as follows [12]:

- Decentralized collaboration: Emphasizing peer-to-peer interactions.
- Tool-oriented approach: Focusing on the development of a robust set of tools.
- Multilingual support: Enabling compatibility with multiple programming languages.
- Thin design: Prioritizing a streamlined framework.
- Openness and freedom: Being freely available and based on open-source principles.

To the best of our knowledge, no existing framework encompasses this specific set of design principles. This section aims to delve into these philosophies, elucidating how they have profoundly influenced the design and implementation of ROS [12].

3.3 Preliminaries

3.3.1 ROS-Graph

3.3.2 Roscore

3.3.3 catkin, Workspaces, and ROS Packages

3.4 ROS Communication

3.4.1 Publishers-Subscribers

3.4.2 Services

3.4.3 Actions

3.5 MoveIt! [17]

MoveIt![3] serves as the primary software framework within the Robot Operating System (ROS) for motion planning and mobile manipulation. It has garnered acclaim for its seamless integration with various robotic platforms, including the PR2 [19], Robonaut [1], and DARPA's Atlas robot. MoveIt! is primarily coded in C++, augmented by Python bindings to facilitate higher-level scripting. Embracing the fundamental principle of software reuse, advocated for in the realm of robotics [9], MoveIt! adopts an agnostic approach towards robotic frameworks, such as ROS. This approach entails a formal separation between its core functionality and framework-specific elements, ensuring flexibility and adaptability, especially in inter-component communication.

By default, MoveIt! leverages the core ROS build and messaging systems. To facilitate effortless component swapping, MoveIt! extensively employs plugins across its functionality spectrum. This includes motion planning plugins (currently utilizing OMPL), collision detection (presently incorporating the Fast Collision Library (FCL) [10]), and kinematics plugins (employing the OROCOS Kinematics and Dynamics Library (KDL) [14] for both forward and inverse kinematics, accommodating generic arms alongside custom plugins).

MoveIt!'s principal application domain lies in manipulation, encompassing both stationary and mobile scenarios, across industrial, commercial, and research settings. For a more comprehensive exploration of MoveIt!, interested readers are encouraged to refer to [2].

4 Basics of Powder Dosing

4.1 Affecting Parameters

The credits for the image above the chapter title go to: wallpaperflare, https://www.wallpaperflare.com/flour-in-a-jar-bake-bakery-baking-powder-wheat-seasoning-wallpaper-agdww

Experimental Set-up, Methodology, and Results

Experimental Set-up 5

5.1 Frame

5.2 Crucibles

5.3 Balance

5.3.1 Hardware

5.3.2 Software

5.4 Niryo-Ned2

5.4.1 Hardware Configurations

5.4.2 Software Tools

5.5 Precision Validation

5.6 Vibration Motor

Methodology 6

6.1 Sequence Logic

6.2 Each State of the State Diagram

7 Evaluation & Results

7.1 Evaluation

Appendix

A

FX_ROS.py Library in Python

```python
#!/usr/bin/env python
import tf
import time
import sys

import numpy as np
import matplotlib.pyplot as plt

import rospy
#from ActionClient import ActionClient
from sensor_msgs.msg import JointState
from niryo_robot_arm_commander.srv import GetFK, GetFKRequest,
     GetJointLimits, JogShift, JogShiftRequest, JogShiftResponse
#from niryo_robot_msgs.srv import SetBool, SetBoolRequest, SetInt,
     SetIntRequest, Trigger

#from niryo_robot_arm_commander.msg import ArmMoveCommand, RobotMoveGoal,
     RobotMoveAction

from moveit_msgs.srv import GetPositionFK, GetPositionIK

from std_msgs.msg import Header
from moveit_msgs.msg import RobotState as RobotStateMoveIt

from geometry_msgs.msg import Pose
import geometry_msgs
from niryo_robot_msgs.msg import RobotState
import moveit_commander
import moveit_msgs.msg
import actionlib

#robot = moveit_commander.RobotCommander()
#scene = moveit_commander.PlanningSceneInterface()
#global arm
def Connect_to_arm():
    global arm
    try:
        arm = moveit_commander.move_group.MoveGroupCommander("arm")
    except:
        raise RuntimeError

def Call_Aservice(service_name, type, request_name=None, req_args=None):
    """Call a ROS service.

    Parameters:
    ........................
    service_name: str
    type: srv
    request_name: None (srv)
    req_args: None (dictionary) ex. {'positon': 210, 'id': 11, 'value':
     False}
    should_return ?: None (int) >> is set to 1, if you want to return the
     response of the service.

    Returns:
```

```python
    ........................
    If should_return is set to 1, the function is going to return the
     response of the service.
    Otherwise, the function should only call the service to do a certain
     action with no return.
    """
    try:
        rospy.wait_for_service(service_name, 2)
    except (rospy.ServiceException, rospy.ROSException) as e:
        rospy.logerr("Timeout and the Service was not available : " + str(
     e))
        return RobotState()

    try:
        service_call = rospy.ServiceProxy(service_name, type)

        if request_name == None:
            response = service_call()
        else:
            request = request_name()
            for key, value in req_args.items():
                #print("f{key} = {value}")
                method = setattr(request, key, value)
            response = service_call(request)

    except rospy.ServiceException as e:
        rospy.logerr("Falied to call the Service: " + str(e))
        return 0

    return response

def Subscribe(topic_name, type, msg_args):
    """Subscribe to a certain topic.

    Parameters:
    ........................
    topic_name: str
    type: srv
    msg_args: list >> list of strings, which contains the arguments that
     we need to read from the topic.

    Returns:
    ........................
    Return a list of the read values from each argument.
    If we have only one argument, it returns the value of this argument
     only, not a list.
    """

    #rospy.init_node('FX_ROS_Subscriber')

    try:
        msg = rospy.wait_for_message(topic_name, type, 2)
    except:
        rospy.logerr("Timeout and the Topic Did not recieve any messages")
        return 0


    value = []

    if len(msg_args) == 1:
        value = getattr(msg, msg_args[0])
    else:
        for i in msg_args:
            value.append(getattr(msg, i))

    return value

```

```
def Get_joints():
    """return a tuple of 6 values for each joint from 1 till 6"""

    joints_values = Subscribe('/joint_states', JointState, ["position"])

    return joints_values

def get_pose():
    """Gets the pose values from the robot_state topic.
    Return:
    ..................................
    a list of two dictionaries, the first is positions (x,y,z),
    whereas the second is the rpy (roll, pitch, yaw)
    """

    return Subscribe('/niryo_robot/robot_state', RobotState, ['position',
     'rpy'])

def get_pose_list():
    """Use get_pose() function to get the pose, and turn it into a list.
    Return:
    ............................
    A list of floats >>> [x, y, z, roll, pitch, yaw]
    """

    pose = get_pose()
    position = pose[0]
    rpy = pose[1]

    return [position.x, position.y, position.z, rpy.roll, rpy.pitch, rpy.
     yaw]

def Get_FK_Niryo(joints):
    """Give the the joints' values to the forward kinematics service
    provided by Niryo, and get the pose coordinations.
    """
    fk_service = '/niryo_robot/kinematics/forward'
    return Call_Aservice(fk_service, GetFK, GetFKRequest, {'joints':joints
     }, should_return=1).pose

def FK_Moveit(joints):
    """Get Forward Kinematics from the MoveIt service directly after
     giving joints
    :param  joints
    :type   joints: list of joints values
    :return: A Pose state object
    @example of a return

position:
  x: 0.278076372862
  y: 0.101870353599
  z: 0.425462888681
orientation:
  x: 0.0257527874589
  y: 0.0122083384395
  z: 0.175399274203
  w: 0.984084775322

    """
    rospy.wait_for_service('compute_fk', 2)
    moveit_fk = rospy.ServiceProxy('compute_fk', GetPositionFK)

    fk_link = ['base_link', 'tool_link']
    header = Header(0, rospy.Time.now(), "world")
    rs = RobotStateMoveIt()

    rs.joint_state.name = ['joint_1', 'joint_2', 'joint_3', 'joint_4', '
```

```
     joint_5', 'joint_6']
    rs.joint_state.position = joints

    reponse = moveit_fk(header, fk_link, rs)

    return reponse.pose_stamped[1].pose


def Jog_shift(joints_or_pose, axis, value):
    """Use the service jog_shift_commander to shift one axis.
    Paramters:
    ........................
    joints_or_pose: int >>> 1 for joints_shift, and 2 for pose_shift
    axis: int >>> (1,2,3,4,5,6) = (x,y,z,roll,pitch,yaw)
    value: float >> the value for which you want to shift the Jog axis.

    Returns: None
    ........................
    """

    axis -= 1
    name = "/niryo_robot/jog_interface/jog_shift_commander"
    shift_values = [0, 0, 0, 0, 0, 0]
    shift_values[axis] = value

    req_arg = {'cmd': joints_or_pose, 'shift_values': shift_values}

    Call_Aservice(name, JogShift, JogShiftRequest, req_arg)

def Move_pose_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ........................
    * axis: str -> (x, y, z, roll, pitch, or yaw)
    * new: float -> The new coordination you want to give to a certain
     axis.
          "new" will always overwrite the value of the axis.
    * add: float -> the value in meters or radians you want to add to a
     certain axis.
    * arm_speed: float (optional) -> between 0 and 1. (0,1]
    Returns: None
    ........................
    """
    FK = get_pose()
    axises = ['x','y','z']

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_current = FK[0]

    rpy_current = FK[1]

    if add:
        if axis.lower() in axises:
            current_value = getattr(p_current, axis)
            setattr(p_current, axis, current_value+add)
        else:
            current_value = getattr(rpy_current, axis)
            setattr(rpy_current, axis, current_value+add)
    if new:
        if axis.lower() in axises:
            setattr(p_current, axis, new)
        else:
            setattr(rpy_current, axis, new)
```

```


    p_goal.x = p_current.x
    p_goal.y = p_current.y
    p_goal.z = p_current.z

    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations.
     quaternion_from_euler(rpy_current.roll,rpy_current.pitch,rpy_current.
     yaw)

    arm.set_pose_target(pose)

    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()

def Move_to_pose(pose_values, arm_speed=None):
    """Move to a given pose values.
    Parameters:
    ........................

    pose_values: list or tuble -> [x, y, z, roll, pitch, yaw]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """

    pose = Pose()
    p_goal = pose.position
    orn_goal = pose.orientation

    p_goal.x = pose_values[0]
    p_goal.y = pose_values[1]
    p_goal.z = pose_values[2]

    roll = pose_values[3]
    pitch = pose_values[4]
    yaw  = pose_values[5]

    orn_goal.x, orn_goal.y, orn_goal.z, orn_goal.w = tf.transformations.
     quaternion_from_euler(roll,pitch,yaw)

    #arm.set_goal_tolerance(0.001)
    if arm_speed:
        set_speed(arm_speed)
    arm.set_pose_target(pose)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()

def move_to_joints(joints, arm_speed=None):
    """Move to a given joint values.
    Parameters:
    ........................

    joints: list or tuble -> [joint1, joint2, joint3, joint4, joint5,
     joint6]
    arm_speed: float (optional) -> between 0 and 1. (0,1]
    """
    joints_limits = Get_Joints_limits()

    for i in range(6):
        if joints_limits.joint_limits[i].max < joints[i] or joints[i] <
     joints_limits.joint_limits[i].min:
            print("Joint{} = {}, which is out of limit!".format(i+1,
```

```
joints[i]))
            print("Joint{} can not be more than {} neither less than {}".
    format(i+1, joints_limits.joint_limits[i].max, joints_limits.
    joint_limits[i].min))
            return
        else:
            pass

    #arm.set_joint_value_target(joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(joints, wait=True)

    arm.stop()

def Move_joint_axis(axis, new=None, add=None, arm_speed=None):
    """You should either put a value to add or new, not both.

    Parameters:
    ........................
    * axis: int -> the number of the joint that you want to move

    * new: float -> The new coordination you want to give to a joint (axis
    ).
           "new" will always overright the value of the axis.
    * add: float -> the value in meters change in a certain joint (axis).
    * arm_speed: float (optional) -> between 0 and 1. (0,1]

    Returns: None
    ........................
    """
    moving_joints = list(Get_joints())

    if new:
        moving_joints[axis-1] = new
    elif add:
        moving_joints[axis-1] += add

    joints_limits = Get_Joints_limits()

    if joints_limits.joint_limits[axis-1].max < moving_joints[axis-1] or
    moving_joints[axis-1] < joints_limits.joint_limits[axis-1].min:
        print("The joint{} can not be more than {} neither less than {}".
    format(axis, joints_limits.joint_limits[axis-1].max, joints_limits.
    joint_limits[axis-1].min))
        return 0
    else:
        pass

    arm.set_joint_value_target(moving_joints)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(moving_joints, wait=True)

    arm.stop()

def Get_Joints_limits():
    """Getting the limits for each joint.

    You can get any joint limits as following:

    Get_Joints_limits().joint_limits[0 - 5].max (float)
    Get_Joints_limits().joint_limits[0 - 5].min (float)
    Get_Joints_limits().joint_limits[0 - 5].name (str)

    Where 0 for (joint 1), and 5 for (joint 6)
    max, min, or name would give the maximum, minimum, or name of the
```

```
     indicated joint.
    """

    joints_limits = Call_Aservice('/niryo_robot_arm_commander/
     get_joints_limit', GetJointLimits)
    return joints_limits

def set_speed(speed):
    """Set a scaling factor for optionally reducing the maximum joint
     velocity. Allowed values are in (0,1]."""
    arm.set_max_velocity_scaling_factor(speed)

def wait(duration):
    """wait for a certain time.

    :param duration: duration in seconds
    :type duration: float
    :rtype: None
    """
    time.sleep(duration)

def move_with_action(pose):
    """Still under development"""

    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('simple_action', anonymous=True)

    robot_arm = moveit_commander.move_group.MoveGroupCommander("arm")

    robot_client = actionlib.SimpleActionClient('execute_trajectory',
     moveit_msgs.msg.ExecuteTrajectoryAction)
    robot_client.wait_for_server()
    #rospy.loginfo('Execute Trajectory server is available for robot')

    robot_arm.set_pose_target(pose)
    #robot_arm.set_pose_target([0.29537095654868956, 4.675568598554573e
     -05, 0.4286678926923855, 0.0017192879795506913,
     0.0014037282477544944, 0.0001612035813676269 3])
    robot_plan_home = robot_arm.plan()

    robot_goal = moveit_msgs.msg.ExecuteTrajectoryGoal()
    robot_goal.trajectory = robot_plan_home

    robot_client.send_goal(robot_goal)
    robot_client.wait_for_result()
    robot_arm.stop()

def move_pose_orn(pose, arm_speed=None):
    """Move to a given pose values, but with orientation not rpy.

    Parameters:
    ........................
    * pose: A Pose state object

example of the pose state object that should be given:
    ========================
    position:
        x: 0.278076372862
        y: 0.101870353599
        z: 0.425462888681
    orientation:
        x: 0.0257527874589
        y: 0.0122083384395
        z: 0.175399274203
        w: 0.984084775322
    ========================
    """
```

```python

    arm.set_pose_target(pose)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)

    arm.stop()
    arm.clear_pose_targets()

def move_to_named_pos(position_name, arm_speed=None):
    """Avalible names:
    - 'resting'
    - 'straight_forward'
    - 'straight_up'
    """
    arm.set_named_target(position_name)
    if arm_speed:
        set_speed(arm_speed)
    arm.go(wait=True)
```

Bibliography

Here are the references in citation order.

[1] Robert O Ambrose et al. 'Robonaut: NASA's space humanoid'. In: *IEEE Intelligent Systems and Their Applications* 15.4 (2000), pp. 57–63.

[2] Tessa Brazda. *NASA Robonaut first generation*. Accessed. 2023.

[3] David Coleman et al. 'Reducing the barrier to entry of complex robotic software: a moveit! case study'. In: *arXiv preprint arXiv:1404.3785* (2014).

[4] Farbod Fahimi. *Autonomous robots*. Springer, 2009.

[5] Intel. *Types of Robots: How Robotics Technologies Are Shaping Today's World*. Accessed. 2023.

[6] Lentin Joseph and Aleena Johny. 'Getting Started with Ubuntu Linux for Robotics'. In: *Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy*. Springer, 2022, pp. 1–52.

[7] Charles C Kemp et al. 'Humanoids'. In: *experimental psychology* 56 (2009), pp. 1–3.

[8] Kevin M Lynch and Frank C Park. *Modern robotics*. Cambridge University Press, 2017.

[9] Alexei Makarenko, Alex Brooks, and Tobias Kaupp. 'On the benefits of making robotic software frameworks thin'. In: *International Conference on Intelligent Robots and Systems-Workshop for Measures and Procedures for the Evaluation of Robot Architectures and Middleware at IROS'07*. Vol. 2. 2007.

[10] Jia Pan, Sachin Chitta, and Dinesh Manocha. 'FCL: A general purpose library for collision and proximity queries'. In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 3859–3866.

[11] Michael Peshkin and J Edward Colgate. 'Cobots'. In: *Industrial Robot: An International Journal* 26.5 (1999), pp. 335–341.

[12] Morgan Quigley et al. 'ROS: an open-source Robot Operating System'. In: *ICRA workshop on open source software*. Vol. 3. Kobe, Japan. 2009, p. 5.

[13] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[14] Ruben Smits, H Bruyninckx, and E Aertbeliën. *KDL: Kinematics and dynamics library*. `https://www.orocos.org/kdl`. Accessed. 2011.

[15] Andrew Spielberg et al. 'Functional co-optimization of articulated robots'. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2017, pp. 5035–5042.

[16] Adam A Stokes et al. 'A hybrid combining hard and soft robots'. In: *Soft Robotics* 1.1 (2014), pp. 70–74.

[17] Ioan A. Sucan and Sachin Chitta. *MoveIt*. `https://moveit.ros.org/`. Accessed. 2013.

[18] Günter Ullrich et al. 'Automated guided vehicle systems'. In: *Springer-Verlag Berlin Heidelberg. doi* 10 (2015), pp. 978–3.

[19] Keenan A Wyrobek et al. 'Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot'. In: *2008 IEEE International Conference on Robotics and Automation*. IEEE. 2008, pp. 2165–2170.

Notation

The next list describes several symbols that will be later used within the body of the document.

c Speed of light in a vacuum inertial frame

h Planck constant

Greek Letters with Pronunciations

Character	Name	Character	Name
α	alpha *AL-fuh*	ν	nu *NEW*
β	beta *BAY-tuh*	ξ, Ξ	xi *KSIGH*
γ, Γ	gamma *GAM-muh*	o	omicron *OM-uh-CRON*
δ, Δ	delta *DEL-tuh*	π, Π	pi *PIE*
ϵ	epsilon *EP-suh-lon*	ρ	rho *ROW*
ζ	zeta *ZAY-tuh*	σ, Σ	sigma *SIG-muh*
η	eta *AY-tuh*	τ	tau *TOW (as in cow)*
θ, Θ	theta *THAY-tuh*	υ, Υ	upsilon *OOP-suh-LON*
ι	iota *eye-OH-tuh*	ϕ, Φ	phi *FEE, or FI (as in hi)*
κ	kappa *KAP-uh*	χ	chi *KI (as in hi)*
λ, Λ	lambda *LAM-duh*	ψ, Ψ	psi *SIGH, or PSIGH*
μ	mu *MEW*	ω, Ω	omega *oh-MAY-guh*

Capitals shown are the ones that differ from Roman capitals.

Alphabetical Index